

Summer 2014

# Automated snort signature generation

Brandon Rice

*James Madison University*

Follow this and additional works at: <https://commons.lib.jmu.edu/master201019>



Part of the [Computer Sciences Commons](#)

---

## Recommended Citation

Rice, Brandon, "Automated snort signature generation" (2014). *Masters Theses*. 301.  
<https://commons.lib.jmu.edu/master201019/301>

This Thesis is brought to you for free and open access by the The Graduate School at JMU Scholarly Commons. It has been accepted for inclusion in Masters Theses by an authorized administrator of JMU Scholarly Commons. For more information, please contact [dc\\_admin@jmu.edu](mailto:dc_admin@jmu.edu).

**Automated Snort Signature Generation**

Brandon Rice

A thesis submitted to the Graduate Faculty of

**JAMES MADISON UNIVERSITY**

In

Partial Fulfillment of the Requirements

for the degree of

Masters of Science

Computer Science

August 2014

## **Dedication**

*To my wife, Heather, for supporting me and encouraging me to pursue my dreams.  
Without your unwavering support this would not have been possible. I don't say it enough  
but "Thank you and I love you."*

## **Acknowledgements**

To JMU:

Knowledge is a gift worth giving and one I will cherish always.

To Dr. Brett Tjaden and Dr. M. Hossain Heydari:

Your leadership and guidance was a key factor in my accomplishments at JMU. Thank you for your time and the opportunities you have provided.

To the SMART Program:

I never imagined I would be selected and in such short time I have nearly completed my degree and will soon enter the work force. I am glad I dreamed big and applied to the program. Without your support I would not be here today.

To my mother and father:

I truly cherish the time and love you have invested in me. I hope that through my accomplishments, big and small, that you can take some measure of pride.

To my daughters:

Lorin and Sydney you are the loves of my life. May my joy be your joy and may you take pride in my accomplishments.

## Table of Contents

Automated Snort Signature Generation .....	i
Dedication .....	ii
Acknowledgements .....	iii
Abstract .....	ix
Introduction .....	1
Intrusion Prevention / Intrusion Detection System .....	2
Snort .....	3
Snort Rule Syntax .....	4
Snort Configuration .....	8
Snort Rules Files .....	9
Packet Analysis .....	10
Python .....	11
File Formats .....	12
Related Work .....	14
Automatic Signature Generation .....	14
Network Intrusion Detection Systems Using Random Forests Algorithm .....	15
An Adaptive Automatically Tuning Intrusion Detection System .....	17

Stochastic Tools for Network Security: Anonymity Protocol Analysis and Network	
Intrusion Detection.....	18
A set of approaches to evaluate and address the accuracy problem in intrusion	
detection systems .....	20
Methods for Speculatively Bootstrapping Better Intrusion Detection System	
Performance .....	21
My Approach .....	24
Functions.....	25
Experiments & Results .....	33
Lab Configuration.....	33
File Generation.....	35
Signature Matching.....	36
Signature Length.....	47
False Positives.....	50
Signature Validity.....	53
Random Distribution.....	53
Conclusion & Future Work .....	56
Signature Generation .....	57
Service.....	57
Script.....	58

Additional Parameters.....	58
Integration.....	59
GUI.....	59
Noise Traffic .....	60
Summary.....	60
Appendix A: Source Code – autoSnortSig.py.....	62
Appendix B: Supplementary Code .....	73
printHex.py .....	73
genText.py.....	73
randCheck.py .....	73
Bibliography .....	XII

## **List of Figures:**

<b>Figure 1:</b> Firewall.....	2
<b>Figure 2:</b> Example Run .....	25
<b>Figure 3:</b> Complex Run.....	25
<b>Figure 4:</b> Example Snort Run .....	34
<b>Figure 5:</b> Wireshark Search.....	35
<b>Figure 6:</b> Percentage of file length.....	55

## **List of Tables:**

<b>Table 1:</b> Snort Rules .....	5
<b>Table 2:</b> IP Packet.....	10
<b>Table 3:</b> Content Matching.....	31
<b>Table 4:</b> Text Files .....	38
<b>Table 5:</b> Word Files .....	39
<b>Table 6:</b> 1997 Word Files .....	40
<b>Table 7:</b> Excel Files .....	41
<b>Table 8:</b> 1997 Excel Files .....	42
<b>Table 9:</b> Executable Files .....	43
<b>Table 10:</b> PDF Files.....	44
<b>Table 11:</b> Word File Comparison.....	46
<b>Table 12:</b> False Positives .....	48
<b>Table 13:</b> False Positives Rate.....	50
<b>Table 14:</b> Packet Capture False Positives.....	51



<b>Table 15:</b> File Types False Positives .....	51
<b>Table 16:</b> Large Signature False Positives .....	52

## **Abstract**

Network intrusion systems work on many models, but at their core they rely on algorithms to process data and determine if the network traffic is malicious in nature. Snort is the most widely-used open source network based Intrusion Prevention System / Intrusion Detection System (IPS/IDS) system. It works by comparing network traffic to a list or lists of rules to determine if and what action should be taken. These rules are referred to as signatures, since they are intended to identify a single pattern of network traffic just like a physical signature identifies a single author. I have developed an algorithm that accepts as input any file or a directory and outputs Snort signatures. This action allows a quick turnaround in creating a rule to stop specific information from traversing the network. By using such a tool, Systems Administrators can better protect their environments through custom rule sets. To verify the algorithm, I generated files of various types containing randomized content and parsed them to generate rules. I then used a Snort installation to process the rules and a packet capture containing the files to determine if the rules operated as intended.

Previously, the creation of rules typically was limited to a very small group of experts that focus solely on such tasks. The core of this research is to enable users to easily create a custom Snort installation, in addition to utilizing the default signatures all Snort deployments use. This increases the security of the assets that each site considers valuable and can be used to prevent data breaches that a typical IDS/IPS deployment could not. The algorithm I have developed is a beginning to the process of creating custom rule sets in an automated manner based on the unique content of each user's environment.

## **Introduction**

The goal of my research is to generate Snort rules from any type of file. I designed and implemented an algorithm which takes as input a file or directory and supports optional arguments and outputs a valid rule(s) that will match the contents of the given file or directory. It would have two-fold use: 1. it can simplify the rule generation process and 2. may also be used to automate the rule generation process when combined with a recurring scheduled task. Both of these functions increase the value of a Snort deployment, and they can be used to defend against unwanted traffic. Additionally, this tool would make the content matching feature of Snort available to less experienced administrators, while also reducing the time it would take to quickly deploy a content matching rule. In typical Snort deployments a Snort system will pull rules from a third party source and take action based on these stock rules. Great effort goes into creating stock rules that take action against the latest threats; but even in the best case scenario, the Snort system is limited to using rules designed for a generic environment. By implementing custom rules, in addition to the standard rule baseline, the Snort system can be tailored to the requirements of a unique network environment or to the unique business needs of a deployment.

A security tool such as Snort is most valuable when it is used for its intended purpose and to its fullest extent. By lowering the requirements to utilize the advanced features of Snort, the security tool enables network administrators to more easily deploy a custom defense configuration. This security component is often overlooked, because a custom security configuration allows for a configuration designed to protect those assets valuable to an organization. By limiting Snort and other tools to the standard out-

of-the-box configurations, they are being underutilized. The algorithms and processes developed during this research are designed to change the difficulty of deploying a custom configuration of Snort, as well as allowing for more interesting uses by adding onto the core algorithm.

This project is a means to decentralize the control of a typical Snort installation's security posture. By expanding the number of administrators who can create custom rule sets, the difficulty of extracting valuable information from a network is increased. This can work against attackers and an accidental exposure by a user of the network. By identifying the valuable information and limiting the paths it can take across the network, Snort can be used to prevent exposures and secure sensitive data. The algorithm developed through this project contributes to the field of information security by taking a new approach to a common problem of generating signatures. Through the current and future work the Snort system can be empowered to provide additional protection specific to each user's needs.

### **Intrusion Prevention / Intrusion Detection System**

The majority of network defense devices work in a similar high level manner. They are designed to allow good traffic in without permitting bad traffic in. However, there are various methodologies these machines use to provide this security functionality. In the case of intrusion, IDS/IPS are operating in a set configuration which determines if traffic is good (allowed) or bad (other actions taken).



**Figure 1:** Firewall

Both intrusion prevention and intrusion detection systems work in the same way; intrusion prevention systems have the added ability to prevent certain actions where an intrusion detection system would simply detect the action and generate an alert or a log entry. These systems work based on two models: signature and anomaly, which dictate how they determine if action needs to be taken. In anomaly based systems a baseline of events is established over a period of time, and future events are compared to this baseline to determine if an alert should be generated. This process is often difficult to calibrate, but can be used to detect zero day exploits or any other potentially malicious traffic that falls outside of the set baselines. The difficulty with this type of system is it being sensitive enough to take action when needed without creating an abundance of false positives.

In contrast, a signature based system will analyze actions and compare them to a database of signatures to determine if action should be taken. A common example is an anti-virus solution which will scan files of a system and determine if any match the signature of a malicious file.

## **Snort**

Snort is a signature based network intrusion prevention system / intrusion detection system originally released in 1998; it has grown to be an industry leader in open source security solutions. At the latest count, it has over four million downloads and four hundred thousand registered users; thus making it the most widely deployed intrusion

protection technology in the world<sup>1</sup>. Snort has the ability to perform protocol analysis as well as content matching giving it the ability to detect and act on a large range of potentially malicious activity such as buffer overflows, stealth port scans, CGI attacks, SMB probes and OS fingerprinting attempts<sup>1</sup>. For this project, it is the content matching engine we are looking to utilize when generating a rule based on an input file.

Snort is programmed in the C programming language and supports deployments across both the Windows and Linux operating systems. As an open source initiative, it has extensive support of supplementary projects as well as the integrity of being available for review by any user. Similar to the field of cryptography where the security should only rely on the security of the keys, an open source project does not provide security through obscurity.

Snort signatures are created by analyzing protocols or the content of network traffic. For this project, the content matching ability of Snort will be utilized to create a rule which matches the content of a file and allows the specified action to be taken. This requires an algorithm which parses the file and additional optional parameters to generate an output rule following Snort syntax with content matching features, so that the file is matched while also minimizing the occurrence of false positives.

### **Snort Rule Syntax**

Snort rules follow a set syntax. Consider the example rule below in **Table 1**:

Action	Protocol	Source IP	Source Port	Direction	IP	Destination	Port	Destination	Parameters
alert	tcp	any	any	->	any		22		(msg:"SSH traffic detected";)

**Table 1:** Snort Rules

### Action:

Each rule begins with an action which dictates what the rule will do once all the rule conditions are met. This is a required field and the typical default value is Alert.

- Alert – Generates an alert using the configured alert method and then logs the offending packet
- Log – Only logs the offending packet
- Pass – Ignores the packet
- Activate – Generates an alert and also activates a dynamic rule
- Dynamic – Rules which are activated and then function as a log rule
- Drop – Drops the offending packet and logs it
- Reject – Drops and logs the offending packet and then sends a TCP Reset or ICMP host unreachable response
- Sdrop – Block the packet but do not log it

**Protocol:**

This field is also required and defaults to IP. However, it also supports TCP, UDP, and ICMP values. This list will most likely expand in the future, but for our usage we will focus on the IP protocol as it encompasses the majority of means for moving a file across a network.

**Source IP:**

This field is optional and can be defaulted to “any” which will match all source IP addresses. A single IP address can be provided such as 192.168.1.100 or a CIDR block can be specified such as 192.168.1.0/24 which allows a range of IP addresses to be input.

**Source Port:**

A source port allows the specification of a port or a range of ports that can be used to only process the content of the packets if it matches the provided source port information.

**Direction:**

The directionality operator can be used to limit the direction of traffic that is monitored. The most common scenario is to monitor from source -> destination but the reverse can also be configured using source <- destination and finally the bidirectional operator <-> allowing for traffic to monitored in both directions.

**Destination IP:**

Like the Source IP field this section can be used to limit the scope of a rule or it can be applied to all destinations by using the value “any”.

**Destination Port:**

Like the Source Port field, this field is used to limit the scope of a rule by specifying the destination port or port range to be monitored.



**Parameters:**

This is where the bulk of the rule is contained and is only processed once all the preceding fields are matched. Since the parameters section typically requires the most processing, it is important to limit the scope of rules using the available fields so that the IPS/IDS device can process traffic in real time. A rule that is overly broad such as:

```
alert ip any any <-> any any (msg:"Packet detected");)
```

would result in a vast amount of alerts as every single IP packet would be matched by this rule. The parameters most focused on are the content, message, and SID fields.

**Content:**

Content is used to match the provided rule's content to that of the traffic that matches the rule conditions. An example rule would be:

```
alert ip any any -> any 80 (content:! "GET");)
```

This rule would monitor the HTTP port of a web site and alert the end user when a request that did not contain the ASCII characters "GET". This is because the negation operator was used to search for the absence of instead of the presence of a variable. The content parameter also supports the use of binary data through hexadecimal notation. For example:

```
alert ip any any -> any any (content:"|0A|");)
```

This allows for packets to be parsed at the byte level. The pipe character | is used to specify the contained string to be parsed in its binary equivalent. In this instance, the hexadecimal content 0x0A is equal to the binary characters 00001010. This is the method that this project uses as not all files contain ASCII printable characters. Instead, the files

are parsed for a binary string represented as hexadecimal characters to be used with the content parameter.

### **Message:**

The message parameter is specified using the keyword “msg” and is used to append a string to an alert or log so that it is easier to know what rule was triggered. By simply looking at rules it is not always obvious what the intent is. This creates alerts and log entries which are more actionable and subsequently more valuable. For example, adding the message parameter to a previous rule:

```
alert ip any any -> any 80 (content:!\"GET\"; msg:\"Non-get web request detected.\");
```

When this alert is triggered, the alert or log entry will contain the message “Non-get web request detected.” which allows the recipient to take any additional necessary action in a timely manner.

### **SID:**

The SID, or Snort ID parameter, is used to give each individual rule a unique identifier, so that it can be referred to easily. This is useful because it allows the triggering rule to be easily be located within the rules file for editing or even temporary disabling (by commenting out the rule). In the absence of a message parameter, the SID parameter is a great tool for pairing an alert or log entry back to the generating rule.

### **Snort Configuration**

All of the Snort configuration is stored within a single file typically named snort.conf. It is here the various settings of a Snort deployment can be customized and additional rule sets added. As a typical rule, file inclusion would look like the below within the snort.conf file:

```
include $RULE_PATH/local.rules
```

This allows a file called `local.rules` to have its contents included in the rule set the next time that Snort is started. The ability to add local customized rules files is important because it allows the third party rule sets to be updated without overwriting local rules. Snort contains a large list of customizable configuration options that each administrator should consider when adapting a deployment. However, these options outside of the inclusion of a rules file are outside of the scope of this project.

### **Snort Rules Files**

Snort rules files follow the typical Python commenting syntax where any line that begins with the `#` character is not processed. This allows for a header that is commented out to be placed at the top of the file; then subsequent lines will contain the rules. Each rule is located on a single line. The file is simply a list of rules to be processed by Snort at runtime. These rule files are where the output of this algorithm should be placed so that on Snort restart the new rule is included in the active rule base. This can be done simply by appending the output of the script to the desired rules file through a command prompt or terminal. Alternatively, the output of the script can be manually added to the rules file. Snort supports the usage of multiple rules files which allows for a separate rules file as needed. This allows a deployment to still subscribe to the standard rules files while creating additional rules files containing their own custom rules.

Snort rules files are parsed at start-up time which means any changes to the static rules files will not immediately take effect. For any changes to be applied, the Snort system must be stopped and restarted. If Snort processes a rule file that is not valid, then it will

exit and not provide network security. For this reason, it is vital that all Snort rules contained within a rules file follow the appropriate syntax.

### Packet Analysis

Snort's content matching system employs a packet level analysis to determine the content of network communications and parse them for signature matches. To better understand this process the structure of an IP packet must be understood.

← 32 bits →				
Version	IHL	Type of Service	Total Length	
Identification			Flags	Fragment Offset
Time-to-live		Protocol	Header checksum	
Source Address				
Destination Address				
Options				
Data (variable length)				

**Table 2:** IP Packet

**Table 2** displays the standard format of an IP packet. The packet head contains the necessary information to route the packet from source to destination and is often updated as the packet transverses routers. The packet header is the portion of the packet which is processed initially to determine if it is a possible match for any of the signatures within Snort. If the packet header matches any rules, then the content of the packet is processed to determine if it matches any of the rules not yet eliminated. Through this process, Snort can process the smaller packet headers and limit the scope of the signatures to be checked. Only once matching rules are found is the packet's contents checked.

Additionally the packet's contents are still checked against all rules even if a previous rule was matched. This allows a single packet to trigger one or more rules. Snort also supports the ability to process fragmented packets where a single payload is spread across multiple packets. Snort will then collect all the fragmented packets and process the content as a single entity preventing the use of data fragmentation as a means to bypass a Snort signature.

Relevant to this work is the ability to use packet capturing software to later be processed by the Snort system. This allows network traffic to be captured, analyzed, and then processed. This is important because it allows data to be permanently created, tested, and then verified manually by viewing the packet capture contents and searching for the generated signature. This also allows later verification of the testing process by another party using the generated network traffic, source file, and algorithm. The ability to duplicate results is vital to creating an algorithm that is dependable and brings value to the end users.

Packet captures are stored within a file with extension .pcap and can be processed by Snort at a later time. This allows a set portion of network traffic to be saved and evaluated by Snort using various rule sets.

## **Python**

When I first approached this problem, I knew that I needed to first select a programming language prior to attempting to outline the algorithm. The criteria I established was a concise language that could operate in the environments where Snort is utilized, fast, and powerful. I examined languages such as C, C++ and Java, but found they were lower level and performance oriented. This algorithm is not burdened with a high workload

since it processes files that are limited in size and processes files in a serial manner. As trade off for a slight performance loss, I used a language with very weak type casting and variables able to adapt to various input provided. This allowed me to focus on the transformation of the input into an acceptable output instead of worrying about pointers or variables with a maximum size.

In addition to its concise and powerful notation, the Python language comes with built-in functionality for reading files and converting data to hex values. This greatly simplified the process, as demonstrated in the code `printHex.py` in Appendix B, of converting a file into a hex string. Once this was done, the task of parsing the file into a signature is much simpler. These built in functions allowed me to utilize known, good code while also keeping the length of my own code to a minimum. By keeping my own code base minimal, it is easier for outside parties to understand and validate contents.

### **File Formats**

In the simplest form, all files are stored in binary format as a string of 1s and 0s. Like most data, it is not very useful until it is given meaning. With computers this is done by giving files types, so that they can be processed according to the contents they are storing. In this way, each string of 1s and 0s can identify that they should be parsed into a picture, an executable, or a document. Even within a specific file purpose, such as a picture, many formats are available to encode the data into a string of 1s and 0s (such as JPG, BMP, and PNG).

This can lead to situations where the format of a file requires that it contain subsections of a certain length. Ideally, any algorithm that created signatures from files would only be concerned with the specific subsections that hold the data unique to a specific file.

However, this would require all file formats be known and each would be processed in a custom manner.

Instead, the algorithm treats all file formats as a string of binary characters to be processed. Assumptions are made to attempt to align the signature generation algorithm with the portion of the files more likely to contain unique data. To do this, the beginning and end of the files are avoided as these are the common locations that a file format would use to store additional information such as the date created or the author. It is also assumed that the file(s) that are to be parsed are in the format they will transverse the network. Any later change such as encryption or compression would result in a change of the files binary content and thus the two formats would require separate signatures.

## **Related Work**

### **Automatic Signature Generation**

In a 2009 paper<sup>2</sup> Alysson Santos of the Federal University of Pernambuco approached the problem of creating, testing, and characterizing signatures for a deep packet inspection (DPI) system in an automated manner. This work simply on topic alone mirrors my work with the primary difference being Santos was working with network traffic, whereas I am working with flat files. This differentiates the necessary algorithm since Santos is generating multiple signatures across a growing data set. In contrast the algorithm I developed takes a set data set defined at runtime and creates signatures on a one-to-one basis. Santos followed a methodology of creating tools which generated signatures, generated patterns, and finally a tool to generate traffic. This allowed her to create an environment that would mimic an actual network, and then test it to see how effectively she was generating signatures.

Santos built upon a previously developed algorithm to parse the generated traffic and identify substrings that were repeating and could be used to classify traffic. Santos ran into difficulty handling large network traffic as it required an exponential amount of memory and computations to parse the file and generate signatures. Despite this, the signature was effective at generating usable signatures, but it did require calibration. The requirement of calibration caused her algorithm to be less automated than she desired. However, it did simplify the signature generation process and allowed for already generated signatures to be tested.



Santos also built a traffic generator which created point to point configured traffic that allowed her to test already generated signatures. This tool is independent of her own signature generation process and could test signatures generated by other means as well as her own. This allowed her to validate the algorithm she was utilizing to create the signatures as well as provide additional value to users of her tools.

Santos' work provided a framework on how to approach my own research. I needed an algorithm that generated signatures as well as a means to test the validity of my results. This lead me to first focus on the Python algorithm which parses the file(s) and outputs the signature. From there, I developed a methodology that allowed me to verify the validity of the results I had obtained. Additionally, I focused on the performance aspects of my algorithm by limiting the total size of the file or file portion to be processed. I was also able to minimize the application's footprint by processing files sequentially, which was not an option for Santos. This is achieved because each file is treated as a unique data set and processed individually. Santos was required to process all traffic as a single entity which is why she had problems with scaling her algorithms.

### **Network Intrusion Detection Systems Using Random Forests Algorithm**

In a 2005 paper<sup>3</sup> Jiong Zhang of Queen's University approached the limitations of rule-based signatures within intrusion detection systems. To do this he approached the concept of a hybrid system that utilized a rules-based system to detect known intrusions and also used an outlier detection algorithm to determine when traffic was outside the norm and possibly malicious. To do this he parsed random forests over training data to establish baselines, and then he detected events that were outliers. This combined approach to

intrusion detection is designed to provide better security by utilizing two technologies that have weaknesses that are offset by each other.

This research is similar in that it wishes to automate the signature generation process and thus improve over the traditional intrusion detection system which operates on rules often created manually. Manual rule creation is a process that requires a high level of knowledge and is time consuming. This is a challenge that Zhang and I both approached using different methodologies.

An interesting process employed by Zhang was to use the misuse detection engine which is rule based to screen data prior to parsing it for outliers. This removed the bulk of intrusions as they will utilize a known attack vector. By doing so, the resulting dataset was relatively clean and allowed for a more accurate baseline to be established. This resulted in baselines which accurately captured the benign traffic and could then be used to better identify outliers. Without this process, the number of intrusions attempted against a system that is publicly accessible leads to baselines that may include intrusion attempts and thus would not be good for establishing a baseline.

The forests algorithm used by Zhang has been applied in other fields and is valid in its ability to predict future events based on a dataset. This allows Zhang to parse a dataset and use it to predict future events. This intrusion prediction data can then be used to better establish rules that protect against malicious traffic that is likely to come. This is very akin to machine learning where a dataset is used to allow an algorithm to learn to make better decisions over time. Applying this concept to intrusion detection just makes sense.

From this work I was able to better understand the limitations of rule based systems such as Snort. Using this knowledge I attempted to also bridge the gap of creating signatures in an automated manner. While Zhang applied his methods to network traffic, the concepts and the pitfalls he worked to overcome are also applicable to flat files.

### **An Adaptive Automatically Tuning Intrusion Detection System**

In 2007, Zhenwei Yu wrote a thesis dissertation<sup>4</sup> for the University of Illinois Chicago regarding an intrusion detection system that eases the calibration challenge typically associated with such devices. When an intrusion detection system is under calibrated it results in a large amount of false positives which can overwhelm an administrator resulting in valid alerts being disregarded or under-investigated. In contrast, a system that is overly tuned will not report intrusions and loses value, because it is not accomplishing its primary task. To overcome this problem, Yu proposed a system that would calibrate itself in an automated manner using user feedback. Primarily, the user would identify false positives as they arose and using this information the system could be calibrated to the user's desired level without the user engaging in direct calibration.

To approach this problem, Yu first had to analyze an intrusion detection model and determine which components would need to be adjusted to calibrate the device. This became complicated because many intrusion detection models are in place including the rules-based system employed by Snort. Yu discovered the best solution was to employ a typical rules-based system and then combine it with a predictive system. It is these predictive systems that are tuned to reduce false positives. Since rules-based systems are designed to detect only certain instances they can do so with a high degree of assurance. From that point the predictive model was used to generate events which the administrator

would then classify as legitimate or as a false positive. This feedback was then used to feed the algorithm which would alter its detection algorithm and then wait for further user input.

Similar to my research, the goal of this paper is to automate a difficult process involving intrusion detection systems. Also, similar to the work completed by Zhang, a hybrid model was employed that built upon the rule-based model employed by Snort. This model goes beyond a simple content checking system and attempts to determine outlier instances that may be malicious. This paper identifies the weaknesses and difficulty of working with a rules-based system and allowed me to focus my work on easing this process and creating a better value from rules-based systems.

### **Stochastic Tools for Network Security: Anonymity Protocol Analysis and Network Intrusion Detection**

In 2012, while at Clemson University, Lu Yu researched anonymity protocol analysis and its impact on network based intrusion detection systems<sup>5</sup>. In specific, Yu analyzed the anonymity protocol The Onion Router (TOR) and how it may be possible to detect a protocol despite it being tunneled through such a system. The ability to detect protocols taken across an anonymity service such as TOR would be a boon for any large network provider, such as an Internet Service Provider (ISP) or an entity tasked with surveying a network.

One such entity tasked with surveying a network is the intrusion detection system. Yu proposed a system that utilized a combination of honeypots and intrusion detection systems to provide a network security scheme that was able to make predictive protocol matching based on past data sets. A key challenge to such a system is the ability to scale

as the network and thus the network traffic grows. For the system to operate it requires the honeypots and intrusion detection systems to be placed throughout the network and then combine the data into a single data set that can be used to make predictive protocol matching and thus attempt to take action against traffic despite an end user using an anonymity service. This is a clear example of exponential growth as the number of data collection devices increases the likelihood of detecting traffic at multiple points increases; thus the total data to be processed grows by a factor greater than one.

At the core of this research is the problem of attempting to identify certain behavior within a data set that has been anonymized. I was able to analyze the methods Lu used to improve my algorithm in order to extract a signature from within a file that could be of any data type. In essence, I had to approach each file without any assumption about the contents of the file since the algorithm was required to process all data types. This mind set is very similar to the process of attempting to extract from anonymized data.

Yu concluded through side channel attacks it was possible to make predictive assumptions about activity being taken by users using anonymity services. This is done by analyzing the delay between each pair of successive packets and then identifying protocols based on the initiators of a particular service. This is done by eavesdropping on the respective communication partners to attempt to ascertain the identity of users.

Finally, Yu proposed a combination of honeypots and host-based intrusion detection systems may be the best approach to limiting the effectiveness of anonymity services. If traffic can be monitored from one end point to another, then it is possible to make certain assumptions and in some cases determine the nature of the communications, despite the anonymity service.

## **A set of approaches to evaluate and address the accuracy problem in intrusion detection systems**

In a 2010 paper Frederic Massicotte of Ottawa-Carleton Institute of Electrical and Computer Engineering wrote a thesis<sup>6</sup> regarding the accuracy of intrusion detection systems. By accuracy, Massicotte was speaking primarily about the false positives that can overwhelm an administrator and distract him or her from the alerts worth investigating. To accomplish this he analyzed the process of testing and evaluating IDS devices to determine if and when they are accurate. Additionally, he looked into intrusion detection systems signatures and how to automatically generate signatures.

One of the key components to Massicotte's research was a model for testing of intrusion detection systems that determined accuracy of the device. As a specific example, Massicotte used the Snort intrusion detection system to illustrate how the process would work in a real-world scenario. This was valuable insight since the Snort program is featured prominently in my research as well. Massicotte tested across three levels (predicate, logic, and state machine levels). He designed test conditions and criteria allowing for meaningful results. This methodology can be applied to other intrusion detection systems to allow for comparison between systems or between different configurations of the same system.

Massicotte also worked on the verification of signatures as a means to improve accuracy. In particular, he worked on signature overlap where one signature can apply to multiple subjects. This was a problem I encountered with certain data types where the algorithm would generate a signature that applied to multiple files. This is far from ideal, because each rule should have a one to one relationship with the content that it is intended to

match. By analyzing the causes and avoidances of these scenarios, Massicotte was able to improve the signatures and subsequently improve the overall accuracy of the system.

Finally, Massicotte worked on a data mining methodology of creating signatures based on training data sets. In particular, he focused on multi-session intrusion detection systems where the generated rules would span multiple data sets and still be applicable. He also outlined his process for selecting a data set for training and the attributes such a data set should possess. This was applicable to my research as I took datasets and attempted to identify the subset of that data which would best create a rule.

Much of Massicotte's work utilized Snort-based examples and results. This insight into Snort was very valuable and related directly to my work. He also focused on the issue of accuracy by attempting to find an automated solution. As with any calibration process on a dynamic system the work is iterative. By automating this process, the accuracy of the system can be improved at each run time, and the technology is kept up-to-date with the best configuration and rules.

## **Methods for Speculatively Bootstrapping Better Intrusion Detection System**

### **Performance**

In 2012 while with the University of New Mexico, Sunny James Fugate researched methods for improving intrusion detection system performance<sup>7</sup>. While much attention is given to false positives and generating rules within the intrusion detection community the impact of both on performance must be considered. As the number of threats grows, the number of rules actively being parsed also grows putting further strain on intrusion detection systems. Simply keeping up with the increase in legitimate traffic can be cumbersome, but these systems must be able to withstand attacks such as flooding or

denial-of-service attacks where the attackers simply try to make the system exceed its capacity.

As was shown in Santos' paper, the problem of scaling within intrusion detections can be overwhelming and limit the ability to effectively protect networks. Fugate focused primarily on the scaling of intrusion detection systems and what can be done to improve that scaling so that intrusion detection systems can continue to meet or exceed the bandwidth requirements networks are currently facing. To do this, Fugate focused on using predictive methodologies to improve system performance using past data sets and results to make decisions with less computational resources being used. By doing this, Fugate theorized that this model would be able to decrease the per signature computational costs and thus allow systems to scale to a larger rule base.

To prove his theory was correct Fugate took two approaches. In the first, an algorithm took predictive action and then used feedback to decrease per signature costs. This process worked on creating thresholds where actions were taken once a threshold was met instead of looking for a 100% match within a rule. The second approach focused on instances where an overburdened intrusion detection system was attempting to function. The standard behavior in this scenario is to drop packets while the system attempts to catch up to the current data waiting to be processed. Fugate demonstrated in some scenarios performance is increased more by simply stopping the processing of rules instead of dropping packets. This theoretically would lower the total error rate and provide better performance for the intrusion detection system.

Once again, this research used the Snort system to show practical implementations of the theoretical models proposed. By focusing on the performance of Snort systems under



stress Fugate was able to highlight some of the pitfalls of intrusion detection systems.

One of these focuses was the per signature computational cost, which is directly applicable to my research as I attempted to automate the signature generation process.

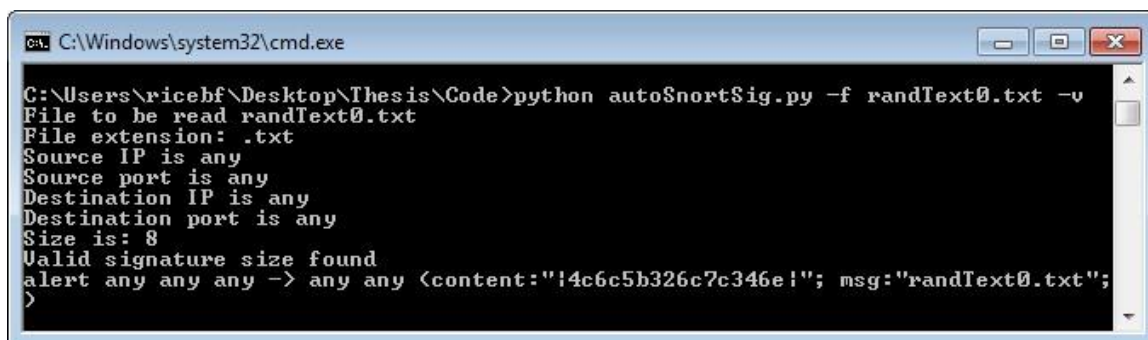
This influenced me to focus on the computational requirements of the signatures my algorithm was generating in an effort to keep the computational costs per rule down. If poor rules are actively used, then intrusion detection systems are vulnerable to flooding or denial or service attacks.

## My Approach

I have created an algorithm which takes as input a minimum of a file or a directory and will output all required Snort syntax to comprise a valid addition to a Snort rules file. The valid arguments which can be provided to the algorithm are:

- f, --file: The path or name of the file to be used.
- a --all: The directory to be parsed. The file and all flags are exclusive.
- s, --source: The source IP and optionally the source port as well. Defaults to the value 'any' if a custom value is not provided at runtime.
- d, --destination: The destination IP and optionally the destination port as well. Defaults to the value 'any' if a custom value is not provided at runtime.
- p, --protocol: Allows specification of a particular protocol to monitor (IP, TCP, UDP or ICMP). Defaults to the value 'any' if a custom value is not provided at runtime.
- l, --length: The signature length in number of bytes. Defaults to a signature length of 8 if a custom size is not provided at runtime.
- v --verbose: Enable verbose output.
- r --random: Randomizes the starting point of the signature generation process. With a fixed starting point, it may be possible for a malicious user to use this to manipulate the signature generation process. By default, random is not enabled.

This encapsulates the basic requirements to generate all the necessary fields for a valid Snort signature. The only required field is the file or directory that is to be parsed. All other fields are optional and will be populated with a default entry if no custom value is provided. This allows for a quick and simple run as illustrated in **Figure 2**:



```

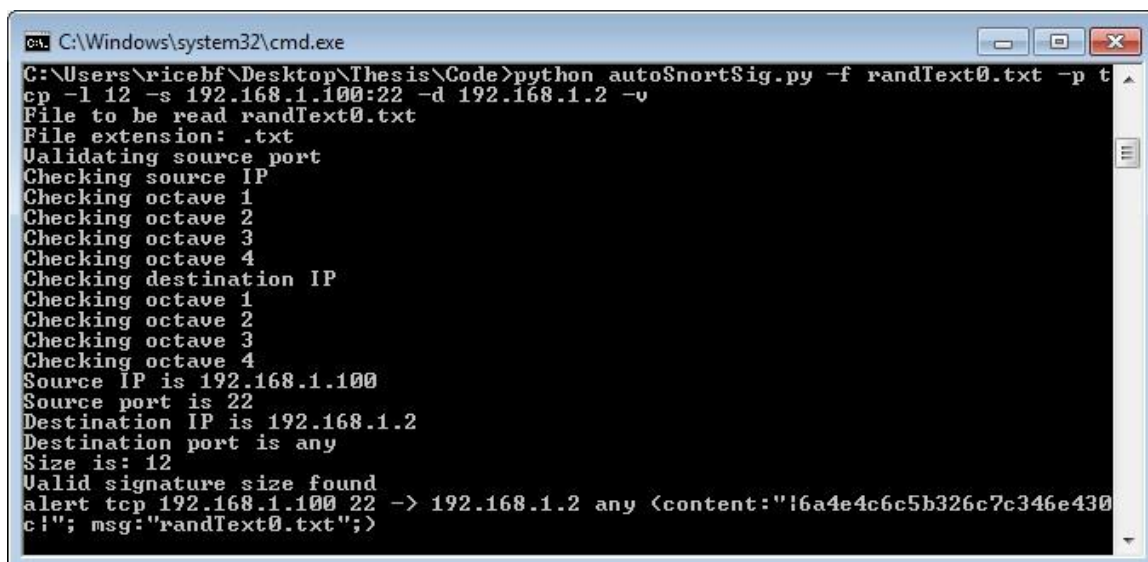
C:\Windows\system32\cmd.exe

C:\Users\ricebf\Desktop\Thesis\Code>python autoSnortSig.py -f randText0.txt -v
File to be read randText0.txt
File extension: .txt
Source IP is any
Source port is any
Destination IP is any
Destination port is any
Size is: 8
Valid signature size found
alert any any any -> any any (content:"!4c6c5b326c7c346e!"; msg:"randText0.txt";
)

```

**Figure 2:** Example Run

However it also allows for very complex rules to be created by using the majority of the flags shown in **Figure 3**.



```

C:\Windows\system32\cmd.exe

C:\Users\ricebf\Desktop\Thesis\Code>python autoSnortSig.py -f randText0.txt -p t
cp -l 12 -s 192.168.1.100:22 -d 192.168.1.2 -v
File to be read randText0.txt
File extension: .txt
Validating source port
Checking source IP
Checking octave 1
Checking octave 2
Checking octave 3
Checking octave 4
Checking destination IP
Checking octave 1
Checking octave 2
Checking octave 3
Checking octave 4
Source IP is 192.168.1.100
Source port is 22
Destination IP is 192.168.1.2
Destination port is any
Size is: 12
Valid signature size found
alert tcp 192.168.1.100 22 -> 192.168.1.2 any (content:"!6a4e4c6c5b326c7c346e430
ci"; msg:"randText0.txt";)

```

**Figure 3:** Complex Run

## Functions

Within the algorithm, many core functions have been split so that they can be called individually. This design process was chosen so that individual functions can be called either through the normal command line method or the code can be integrated with other

works with ease. This additionally allowed for easier trouble shooting and validation as each module could be tested individually.

**main()** – The main function is the driver for the algorithm and works by first reading in the arguments passed at startup time. If the user specified a single file the fileParse function will be called or if a directory is specified the dirParse function will be called.

**fileParse(String)** – The fileParse function calls all the necessary functions and routines to parse a single file and output a single signature. It does so by calling the fileRead function to read the file passed to it; then it validates the arguments using ipCheck, validSize, and validProtocol before finally using createSig and combineSig to update the signature String. Finally, it calls a print function to print out the complete signature.

**dirParse()** – The dirParse function uses the line:

$$files = [f \text{ for } f \text{ in } os.listdir(directory) \text{ if } isfile(join(directory,f)) ]$$

to create a list of files that can then be parsed over one at a time using the fileParse function. To achieve this, the dirParse function loops over the files list and calls the fileParse function providing both the directory and file name to be parsed so that the fileParse function is passed a complete file path.

**vPrint(String)** – This simple function checks if the verbose flag was set by the user. It will only print the passed String if the function has been set by the user. This allows certain informational output to be suppressed which is the default behavior so that the output can be piped into a rules file.

**argRead()** – The argRead function utilizes the argparse functionality built into Python to process all the command line arguments and store them into global variables. It also determines the file extension is to be processed. This functions determines when a

directory is given if it will alter the directory to be in the UNIX format using forward slashes instead of the Windows default back slash which Python interprets as an escape character.

**fileRead()** – fileRead will open the file stored in the global file variable and then save the contents of the file into the global fileString variable as hex values. To limit the processing of very large files, the length of the fileString is limited to 2,621,440 characters or roughly 10mb. This means that files over 10mb will only have the first 10mb processed when generating a signature. This is a compromise between speed and signature accuracy. Processing further into files will result in decreased performance and the first 10mb should be sufficient to create a valid signature.

**ipCheck()** – The ipCheck function begins by parsing the given source and destination which can include both port and IP or simply and IP address. At the command line this would look something like:

```
>python autoSnortSig.py -s 192.168.1.1:80
```

Where a source IP address of 192.168.1.1 and a source port of 80 are passed to the algorithm. These strings have to be parsed first by determining if the source or destination information was passed, secondly, by checking if it contains both a port and IP or simply an IP, and finally by storing the port and IP into global variables if passed or storing the default value. Once completed, the validation functions are called if the global variables are not storing the default value.

**validPort()** – First the port functions checks that the provided String is an integer. If is not, then an error is printed and the program exits. If an integer is detected, it must be in the range of 1-65536 or else an error is printed and the program exits.

**validIP()** – This is a driver function that works by separating the provided IP address into octets and then passing those octets one at a time to the validOctave function. If the IP address provided is not valid, then the function prints an error message and exits the program.

**validOctave()** – As each octave is passed to this function, it verifies that it is an integer in the range of 0-255 and will return True if that is the case. For all other cases it returns False.

**validSize()** – This function checks the user provided signature size, verifies it is an integer, and validates it is an acceptable value. The default value for size is eight, but values as small as four and as large as sixty four are accepted. If an unacceptable value is detected, then an error message is printed and the signature size is set back to the default value eight.

**validProtocol()** – The list of acceptable protocols ("ip","tcp","udp","icmp","any") is stored within a list. Any protocol provided by the user must be contained within the list for it to be accepted. In order for this to be accomplished, the global variable protocol is first converted to lowercase and then the list is checked to see if it contains the global protocol. If it does not, the user input is considered invalid causing an error message and the protocol to be set back to the default value of any.

**createSig()** – This function is the core of the algorithm as it takes the file as a String of hex characters and parses it to generate a substring that will be used to populate the content portion of the rule. It also includes a check that will randomize the start location within the search for a valid substring if the random flag is passed by the user. The standard starting location is the middle of the file or the middle of the first 10mb, should

the file be larger than 10mb. The intent of starting in the middle of the file is to avoid any patterns that are common amongst file types appearing at the beginning or end of a file. The random flag changes that so that the algorithm starts at a random location anywhere from 25% to 75% of the files length. Once a start position is determined, a substring is pulled and then tested for validity. It must contain less than 50% of the hex value “0” to be considered valid. This check is put in place to make sure the substring pulled is not a portion of the file that has been padded with binary “0”. If the substring fails, the validity test the algorithm increases the starting location and pulls another substring. Since this process is designed to avoid padding within files it processes the file from the starting point in one direction so that it can find a point that is outside of the padding. The new substring is tested and the process continues until a valid substring is found or the length of the file is exhausted. If no valid substring is found an error message is printed and the program exits.

**combineSig()** – The combineSig function takes the content generated by createSig and combines it with all the necessary syntax to create a valid rule. This includes parsing the already validated global variables to determine the protocol, source, and destination to be included in the final rule. These elements are all combined in the correct order and necessary grammar to form a complete rule.

When all of these functions are combined the program follows a precise controlled flow. Either a single file is parsed or lists of files are parsed one at a time. Regardless, the same core functions are used to generate the signatures and print them as output.

The creation of the signature focuses on the ability of Snort to content match rules to traffic in three ways using the content tag. The content tag allows for rules to be

compared to traffic using binary, hexadecimal, or ASCII strings. This requires the input file to be converted into one of these formats and then parse the file to choose a particular section to be used as the signature. After research, I discovered Python had the built-in functionality needed to read in files and convert them to all three formats prompting me to implement the algorithm. Additionally, with Python being such a high level language, I was able to implement the project using fewer lines of code which allowing better testing and understanding of the data flow.

I quickly discovered ASCII was not the preferred format because many file types have data stored in what would result in non-printable characters. This prompted complications because the Snort rule required characters to be printed so that they could be included in the resulting rule (which are typically edited with a text editor such as Notepad or gEdit). When choosing between binary and hexadecimal, my findings concluded they were functionally the same, but hexadecimal characters permitted rules with sufficiently large signatures to be generated without creating excessively long rules. For instance, a signature that is 4 bytes long would require 8 hexadecimal characters, but would be composed of 32 binary characters. For this reason, I decided that all signatures would be output in hexadecimal format.

The process of parsing the string of hex characters into a suitable signature was one that had a wide array of challenges. The primary challenge, however, is that each file format is unique and often includes standard header and footer content that is not unique to a specific file. Including this content is unique to a file type and would result in all files of the same type also being identified by the generated rule. To limit this likelihood, the



signature is extracted from the middle of the file attempting to target the area most likely to contain content unique to the inputted file.

Another potential pitfall is in the splitting of bytes. Because a byte is represented using two hexadecimal characters, it is possible for the second half of a byte to be the first hex character in the signature and the first half of a byte to be the last character in the signature. When processed by Snort, this would result in unintended matches. For example, with a signature length of 3 it is possible for the following to occur: (see **Table 3**):

Byte Number	1	2	3	4
File Content	A0	78	3C	FF
Extracted Signature	0	78	3C	F
Signature that will be used	07	83	CF	

**Table 3:** Content Matching

In this example, the last half of the first byte is taken from byte one and the first half of byte four is taken to create a signature that is three bytes long. However, the bytes “07 83 CF” are not contained in the original file content so a bad signature has been created.

To address this problem the hex characters are combined in pairs into an array so that each element of the array represents a single byte of data. This prevents the accidental splitting of bytes and the end result is signatures that are more likely to match the original file content.

The final pitfall discovered was the padding within files which could output a signature whose content was composed either primarily or solely of “00” bytes. While this would result in a good match with the original file contents, it was also overly broad and would

match files it should not. To prevent this, the occurrence of “0” within the generated hex signature is checked. Based on the size of the signature, if the number of “0” characters is too high the signature is discarded and the file is iterated over until a suitable signature is found.

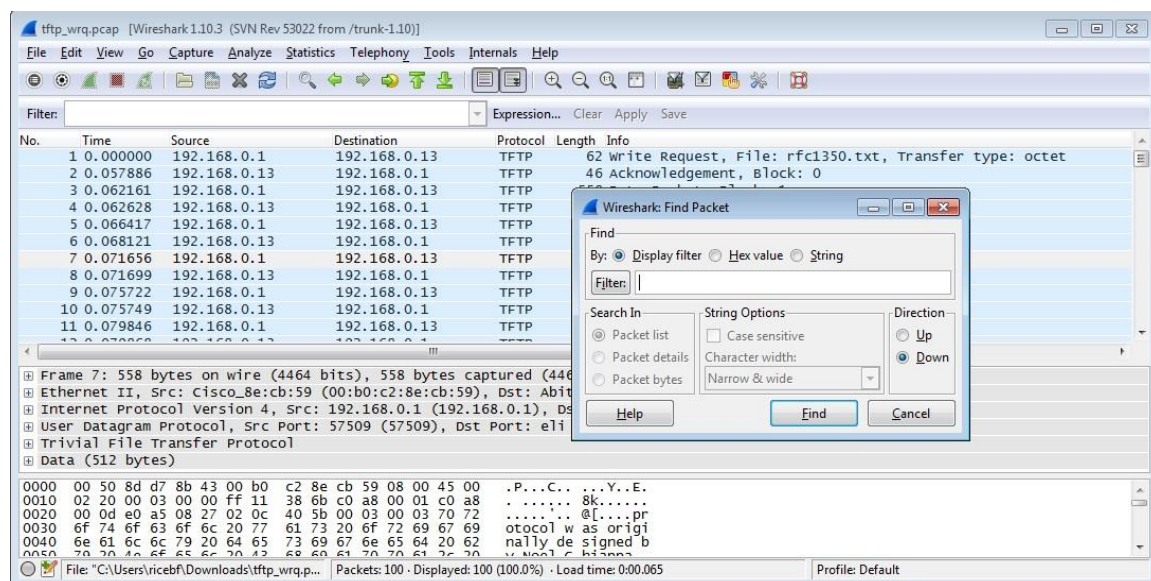
## Experiments & Results

### Lab Configuration

I had two primary requirements of the lab. My first requirement was the ability to run the Snort software and the need to create packet captures of files being moved across the network. The easiest configuration I found that met these goals was a virtual machine running Ubuntu 12.04 LTS. From there the necessary software, Snort, Wireshark, and an FTP server (vsftpd), were installed and configured to a functional level. To capture network traffic containing a specific file I utilized Wireshark while transferring files over the FTP protocol. The resulting packet captures were filtered by port so that only FTP traffic was contained within the created packet capture file. I selected the FTP protocol because of its simplicity in moving files over the network in an unencrypted and uncompressed manner. Moving files in an unencrypted and uncompressed manner is key, because once the content was encrypted or compressed the generated signatures would no longer be applicable. I could then use the created packet capture files as input to Snort using to test for matches to the created rules. By performing an offline analysis of the captured traffic I was able to recreate results and also perform multiple tests across the same packet captures. An example of such a test is shown in **Figure 4**.



Next I took the output of the snort analysis and compared it to the contents of the packet capture file manually using Wireshark to determine if the signature was identifying a portion of the file or simply some portion of the FTP overhead. This process was repeated only for the first few transfers to validate the automated testing mechanisms. An example search is shown in **Figure 5**.



**Figure 5: Wireshark Search**

## File Generation

To test and verify the algorithm I needed to create data sets, in this case files, which the algorithm could parse. Then the generated signatures could be tested. To do this I populated various file formats with sufficient random data in order for a unique signature to be created without creating so much data that the unique format of each file type becomes a non-factor. I decided to focus on the file formats most popular in the typical business network settings, which are Microsoft Office formats, text files, and executables (.exe). The process was straight forward for populating documents, but the executables

were somewhat problematic. Therefore, I relied on publicly available executables to use as a testing data set for the executable file type.

To generate the random ASCII printable characters, I created a short algorithm which printed out 1,000 characters randomly chosen from all possible ASCII printable characters. For reference, the algorithm is included under Appendix B as the file `genText.py`. The output of this algorithm was then piped to various text files to create the text file data sets. This was not possible for more advanced Microsoft Office file types, so instead I copied and pasted the output of the script into the files manually. For comparison, the similar Office files such as `.docx` and `.doc` are using the same random string of printable characters. Once the requisite files were created, then I was able to parse them with the signature generating algorithm and log the results for future testing with Snort.

The quantity of 1,000 random ASCII characters was chosen to attempt to highlight the challenges of finding unique content within various file types. In the simplest example of a text file the chances of finding a substring of the 1,000 ASCII characters within a 1KB text file is very high. In other formats which contain a larger amount of formatting and configuration information the chances of finding the unique content is much lower.

### **Signature Matching**

To determine if the generated signatures were valid I needed to upload the generated signature to a rules file, restart Snort, and then have Snort process a packet capture file that included the file being transferred. I used the default signature size with no additional options (such as protocol or port) so that the rule would search simply for the presence of the file signature instead of searching only a subset of the packet capture file that met the

rule requirements. While this is not best practice in a real world application, the goal of this test was to determine the validity of the content portion of the rule so removing all other possible causes of a signature mismatch was needed. For each file type 10 files were tested to establish a baseline of success / failure for that particular data type. While this is not a large enough quantity to determine if the algorithm works against all files of a particular data type, it does allow for a reasonable determination of success when processing a data type. The following file types were tested with included results as depicted in Tables 4-10.

Text Files	Signature Generated	Results	File Size
<b>randText0.txt</b>	alert any any any -> any any (content:" 4c6c5b326c7c346e ";)	Match	1KB
<b>randText1.txt</b>	alert any any any -> any any (content:" 2031577929200d0a ";)	Match	1KB
<b>randText2.txt</b>	alert any any any -> any any (content:" 5b656c2e2371555b ";)	Match	1KB
<b>randText3.txt</b>	alert any any any -> any any (content:" 09553e56557b694f ";)	Match	1KB
<b>randText4.txt</b>	alert any any any -> any any (content:" 2a470b78792c366d ";)	Match	1KB
<b>randText5.txt</b>	alert any any any -> any any (content:" 6d0b6b39300d614b ";)	Match	1KB
<b>randText6.txt</b>	alert any any any -> any any (content:" 3b59432e2756617e ";)	Match	1KB
<b>randText7.txt</b>	alert any any any -> any any (content:" 682f7b7576657039 ";)	Match	1KB
<b>randText8.txt</b>	alert any any any -> any any (content:" 656c567473357155 ";)	Match	1KB
<b>randText9.txt</b>	alert any any any -> any any (content:" 6747247e0d2e5832 ";)	Match	1KB

**Table 4:** Text Files



Docx Files	Signature Generated	Results	File Size
<b>randText0.docx</b>	alert any any any -> any any (content:" ca5d7399e4b6afab ";)	Match	15KB
<b>randText1.docx</b>	alert any any any -> any any (content:" 2db7e623ec03acd8 ";)	Match	15KB
<b>randText2.docx</b>	alert any any any -> any any (content:" 8dbdf066afd1fc51 ";)	Match	15KB
<b>randText3.docx</b>	alert any any any -> any any (content:" f2ef0bec98317471 ";)	Match	15KB
<b>randText4.docx</b>	alert any any any -> any any (content:" 9a7fadbc29cd4d60 ";)	Match	15KB
<b>randText5.docx</b>	alert any any any -> any any (content:" 93723e8f0aa79894 ";)	Match	15KB
<b>randText6.docx</b>	alert any any any -> any any (content:" a1ee4d8c36203371 ";)	Match	15KB
<b>randText7.docx</b>	alert any any any -> any any (content:" 0796c938ec0d9335 ";)	Match	15KB
<b>randText8.docx</b>	alert any any any -> any any (content:" ccb2f3243b9f65b8 ";)	Match	15KB
<b>randText9.docx</b>	alert any any any -> any any (content:" 72642f7765625365 ";)	Match	15KB

**Table 5:** Word Files

Doc Files	Signature Generated	Results	File Size
<b>randText0.doc</b>	alert any any any -> any any (content:" 8643a2c3a74346e7 ";)	Match	23KB
<b>randText1.doc</b>	alert any any any -> any any (content:" 0c3258726c6f765f ";)	Match	23KB
<b>randText2.doc</b>	alert any any any -> any any (content:" 9822266798222667 ";)	Match	23KB
<b>randText3.doc</b>	alert any any any -> any any (content:" b400818132300000 ";)	Match	23KB
<b>randText4.doc</b>	alert any any any -> any any (content:" b400818132300000 ";)	Match	23KB
<b>randText5.doc</b>	alert any any any -> any any (content:" 9526d50403b7c743 ";)	Match	23KB
<b>randText6.doc</b>	alert any any any -> any any (content:" 530c696f6d3a537d ";)	Match	23KB
<b>randText7.doc</b>	alert any any any -> any any (content:" 9e2226679e222667 ";)	Match	23KB
<b>randText8.doc</b>	alert any any any -> any any (content:" 4433487b705b7a0b ";)	Match	23KB
<b>randText9.doc</b>	alert any any any -> any any (content:" 0000ffffff7fffff ";)	Match	23KB

**Table 6:** 1997 Word Files

Xlsx Files	Signature Generated	Results	File Size
<b>randText0.xlsx</b>	alert any any any -> any any (content:" c6e751cc195885a5 ";)	Match	10KB
<b>randText1.xlsx</b>	alert any any any -> any any (content:" ba9942d311a2d0ad ";)	Match	10KB
<b>randText2.xlsx</b>	alert any any any -> any any (content:" a33b3fd18cba0567 ";)	Match	10KB
<b>randText3.xlsx</b>	alert any any any -> any any (content:" e701ca9951c9aeb6 ";)	Match	10KB
<b>randText4.xlsx</b>	alert any any any -> any any (content:" 5fe7f777e911ddde ";)	Match	10KB
<b>randText5.xlsx</b>	alert any any any -> any any (content:" 9996fd82f31a6dc6 ";)	Match	10KB
<b>randText6.xlsx</b>	alert any any any -> any any (content:" daaddb1a92944129 ";)	Match	10KB
<b>randText7.xlsx</b>	alert any any any -> any any (content:" 71daaddb1a929441 ";)	Match	10KB
<b>randText8.xlsx</b>	alert any any any -> any any (content:" b79f929032e8a537 ";)	Match	10KB
<b>randText9.xlsx</b>	alert any any any -> any any (content:" 9f929032e8a53709 ";)	Match	10KB

**Table 7:** Excel Files

Xls Files	Signature Generated	Results	File Size
<b>randText0.xls</b>	alert any any any -> any any (content:" 2848000050243563 ";)	Match	25KB
<b>randText1.xls</b>	alert any any any -> any any (content:" 0c50212e4044245d ";)	Match	25KB
<b>randText2.xls</b>	alert any any any -> any any (content:" 7e5321206273316a ";)	Match	25KB
<b>randText3.xls</b>	alert any any any -> any any (content:" 4f24575d725d705d ";)	Match	25KB
<b>randText4.xls</b>	alert any any any -> any any (content:" 23e6561393c2e784 ";)	Match	25KB
<b>randText5.xls</b>	alert any any any -> any any (content:" 7422516573556877 ";)	Match	25KB
<b>randText6.xls</b>	alert any any any -> any any (content:" 553a54602a676d2b ";)	Match	25KB
<b>randText7.xls</b>	alert any any any -> any any (content:" 7056596e64537a44 ";)	Match	25KB
<b>randText8.xls</b>	alert any any any -> any any (content:" 5d257b374e424354 ";)	Match	25KB
<b>randText9.xls</b>	alert any any any -> any any (content:" 40422924707d6b79 ";)	Match	25KB

**Table 8:** 1997 Excel Files

Exe Files	Signature Generated	Results	File Size
<b>AIM_Install.exe</b>	alert any any any -> any any (content:" 7176493855395036 ";)	Match	18MB
<b>putty.exe</b>	alert any any any -> any any (content:" 00006a146a506a02 ";)	Match	484KB
<b>FirefoxSetupStub29.0.1.exe</b>	alert any any any -> any any (content:" 2c0c084040b31f41 ";)	Match	277KB
<b>ChromeSetup.exe</b>	alert any any any -> any any (content:" cccccccc8bff558b ";)	Match	898KB
<b>Hearthstone-Setup-enUS.exe</b>	alert any any any -> any any (content:" 465004747251508d ";)	Match	6.7MB
<b>FileZilla_3.7.4.1_win32-setup.exe</b>	alert any any any -> any any (content:" 00002bd1c1fa0503 ";)	Match	4.7MB
<b>Gw2Setup.exe</b>	alert any any any -> any any (content:" 8bf185c0742f85d2 ";)	Match	22.2MB
<b>winscp518setup.exe</b>	alert any any any -> any any (content:" c465842653a29724 ";)	Match	4.9MB
<b>Wireshark-win64-1.10.3.exe</b>	alert any any any -> any any (content:" b663c553c2c6d637 ";)	Match	27.3MB
<b>WinPcap_4_1_3.exe</b>	alert any any any -> any any (content:" 0d0a6129664b0d0a ";)	Match	894KB

**Table 9:** Executable Files

PDF Files	Signature Generated	Results	File Size
<b>randText0.pdf</b>	alert any any any -> any any (content:" 2532fc8a467945e5 ";)	Match	108KB
<b>randText1.pdf</b>	alert any any any -> any any (content:" 2a8ce811da093f7a ";)	Match	111KB
<b>randText2.pdf</b>	alert any any any -> any any (content:" 020fbff42278f819 ";)	Match	111KB
<b>randText3.pdf</b>	alert any any any -> any any (content:" f2e0c47e801378f9 ";)	Match	110KB
<b>randText4.pdf</b>	alert any any any -> any any (content:" 3a46859d63ec00dc ";)	Match	110KB
<b>randText5.pdf</b>	alert any any any -> any any (content:" 800ddae363a89f67 ";)	Match	109KB
<b>randText6.pdf</b>	alert any any any -> any any (content:" 0c82a6e90baff21c ";)	Match	111KB
<b>randText7.pdf</b>	alert any any any -> any any (content:" 41e5296149f35369 ";)	Match	107KB
<b>randText8.pdf</b>	alert any any any -> any any (content:" dcbef3b5dffefa12 ";)	Match	110KB
<b>randText9.pdf</b>	alert any any any -> any any (content:" 90ba89d6c54a5ee5 ";)	Match	107KB

**Table 10:** PDF Files

Tables 4-10 illustrate the generated output when the various files are parsed by the algorithm. Looking at the match results for the one to one test of generated signature to file the algorithm is generating signatures that match well with all the tested file types. However **Table 6** uses the simple format of Word documents .doc, and it shows a potential problem with this methodology. Files 3 and 4 share the same generated signature, despite containing unique randomized content, which will result in both generated rules creating an alert when either file is detected. This is because the algorithm is selecting a signature from a portion of the file that contains information on the formatting of the file instead of the portion that contains the unique random ASCII printable strings. Each file contains 1,000 random ASCII characters which makes it is easy to compare the 1KB text files and the 23KB word files and see that the word files contain an additional 22KB of file format specific content. Since these files are the same size, format, and use the same content generation methods, it makes sense that the algorithm will select a similar portion of each file. However, in a real world application these files sharing the same signature is not advantageous. To overcome this problem the algorithm can either be run using the random flag to select a randomized starting point or a larger signature can be generated. To discover how deep these matching signatures ran, I generated signatures for files 3 and 4 of increasing length until they generated a unique signature.

Length	randText3.doc Signature	randText4.doc Signature
8	alert any any any -> any any (content:" b400818132300000 ");	alert any any any -> any any (content:" b400818132300000 ");

9	alert any any any -> any any (content:" b400b4008181323000 ";)	alert any any any -> any any (content:" 05a005b400b4008181 ";)
10	alert any any any -> any any (content:" a005a005b400b4008181 ";)	alert any any any -> any any (content:" ffa005a005b400b40081 ";)
11	alert any any any -> any any (content:" 00ffffff7ffffff7ffff  ";)	alert any any any -> any any (content:" 05a005b400b40081813230 ";)
12	alert any any any -> any any (content:" a005a005b400b40081813230  ";)	alert any any any -> any any (content:" a005a005b400b40081813230 ";)
13	alert any any any -> any any (content:" 040000ffffff7ffffff7ffff ";)	alert any any any -> any any (content:" ffff7ffffff7ffffff7ffff ";)
14	alert any any any -> any any (content:" 00004b831100f010000800fcfd 21 ";)	alert any any any -> any any (content:" 00a005a005b400b40081813230 00 ";)
15	alert any any any -> any any (content:" 24500000e4040000ffffff7fffff f ";)	alert any any any -> any any (content:" 0f000924500000e4040000ffffff 7f ";)
16	alert any any any -> any any (content:" ffffff7ffffff7ffffff7ffffff7f "; )	alert any any any -> any any (content:" ffffff7ffffff7ffffff7ffffff7f ";)

**Table 11:** Word File Comparison

This leads to an interesting discovery where the generated signatures of length eight through sixteen have a pattern of matches at length eight, twelve, and sixteen. Further



analysis shows that many of the signatures share components with other signatures. This is shown clearly in the signatures for file three where the signature of length 12 contains substrings of the length 8 and 9 signatures. Signature 12 of file 3 contains 75% of signature 8 and it contains 89% of signature 9. This also occurs across files where the signature of length 11 from file 4 is found in its entirety within signature length 12 of file 3. From this I can infer that the two files share similar content at the algorithms starting point which is causing them to create signatures that have commonalities. This is counter intuitive to the idea that a longer signature is more likely to be unique. This illustrates the concept that unique file formats may present unique challenges in creating signatures. Based on the data collected, the algorithm is generating signatures that match with the contents of the file(s) being parsed. However, through analysis of the created packet capture files I found some of the matches did not align with the unique content of the files (such as the words and phrases of a word document). Within the simple text files it was easy to pair the generated signatures to the generated ASCII characters. See **Figure 5** for an example of a Wireshark capture of a text file where the ASCII contents can be read in the bottom right. However the .docx and .xlsx files utilize an XML based format that is spread across multiple files and then compressed to save space. This compression made the process of identifying where the document contents started difficult. Ideally, each supported extension would have a unique handling that generated the signature from the portion of the file that would contain unique content.

### **Signature Length**

I wanted to consult both theoretical and real world scenarios that would lead me to an ideal signature length for the default for the algorithm. For the theoretical scenario, I

wanted to determine the average amount of bytes processed before a positive occurred assuming that the data is random. For a single byte this is simple, because a byte is composed of 8 binary digits and has  $2^8$  or 256 possible values. Therefore, any byte would have a 1 in 256 chance of being the byte we are searching for leading to an average false positive every 256 bytes. Beginning with a signature of two bytes, the formula changes slightly as content of 100 bytes will only have 99 byte pairs (determined by: length – (size – 1)). In order to determine the average false positive rate, we need to determine the chance of any random byte pair being the same and then calculate the length that would result in an average of 1 false positive.

$$rate = 2^k + (length - (size - 1))$$

However the average rates grow exponentially with  $2^k$  so the linear growth of (length – (size – 1)) can be discarded as trivial.

Signature Length	Average Size	Signature Length	Average Size
1	$2^8$ bytes	9	$2^{72}$ bytes
2	$2^{16}$ bytes	10	$2^{80}$ bytes
3	$2^{24}$ bytes	11	$2^{88}$ bytes
4	$2^{32}$ bytes	12	$2^{96}$ bytes
5	$2^{40}$ bytes	13	$2^{104}$ bytes
6	$2^{48}$ bytes	14	$2^{112}$ bytes
7	$2^{56}$ bytes	15	$2^{120}$ bytes
8	$2^{64}$ bytes	16	$2^{128}$ bytes

**Table 12:** False Positives

Based on **Table 12**, the best case signature length would limit the rate of false positives while also keeping the computational requirements to a minimum. A signature of length 8 would produce a positive at the rate of once every 16,384 petabytes. This is sufficiently large enough to deter false positives and is still small enough to conserve computational resources. For this reason the default signature size within the algorithm is 8 bytes.

To determine the real world implementation, I created a ~100mb packet capture file and searched it for signatures of increasing size beginning with 1 byte and ending at 8 bytes. Each signature was a randomly generated hex string with no relation to the randomly generated content of the packet capture. I created the packet capture using a variety of normal activities, such as web browsing and video streaming, totaling in size at 106,915kb. This was all done over an encrypted HTTPS connection which should result in an acceptable level of randomness based on the encryption algorithms used. This test was performed five times with the signatures changing each time and the results averaged in **Table 13** to illustrate the rate of false positives. The rate of positives could then be compared between **Table 12** and **Table 13** to verify that the theoretical results mapped to the real world implementation. A larger packet capture could have provided results into a higher signature length but at a length of four the packet capture would need to be 4,096mb to generate an average of 1 false positive. The adherence to similar rates among the smaller signatures is sufficient to prove that the theoretical and real world implementation support each other:

Signature Size	Occurrences	Average Size
<b>1</b>	432,069.8	253.39 bytes
<b>2</b>	1626.6	67,306.6 bytes

<b>3</b>	6.2	17,244.4 kilobytes
<b>4</b>	.2	547.4 megabytes
<b>5</b>	0	N/A
<b>6</b>	0	N/A
<b>7</b>	0	N/A
<b>8</b>	0	N/A

**Table 13:** False Positives Rate

### False Positives

In this experiment, I took the generated signatures from the signature matching test and compared them against the single large packet capture file (106,915kb) generated in the signature length experiment. By doing so, I wanted to examine the use of the generated signatures in a real world implementation to determine if they were creating an abundance of noise through false alerts. Ideally, none of the signatures would generate an alert since the file they were derived from was not part of the packet capture file.

Additionally, the random packet capture file was generated without transferring any of the file types being tested for so the danger of signatures catching other files of the same type was not addressed with this experiment.

File Type	Results
<b>10 .txt files</b>	No Match
<b>10 .doc files</b>	No Match
<b>10 .docx files</b>	No Match
<b>10 .exe files</b>	No Match

<b>10 .pdf files</b>	No Match
<b>10 .xls files</b>	No Match
<b>10 .xlsx files</b>	No Match

**Table 14:** Packet Capture False Positives

Based on these results, I feel that the signatures are not prone to creating false positives when compared to random data. However, I still needed to test the possibility of signatures being specific to file types and not simply a single file. To do this I created a packet capture file for each file type that consisted of 10 different files of the same type being transferred over FTP. This packet capture was then tested for one of the original signatures found in Tables 4-10. The occurrence of false positives is shown in **Table 15:**

File Type	False Positives
<b>.txt</b>	0
<b>.doc</b>	0
<b>.docx</b>	3
<b>.exe</b>	1
<b>.pdf</b>	0
<b>.xls</b>	1
<b>.xlsx</b>	2

**Table 15:** File Types False Positives

Based on these results, I believe there is concern for false positives of the same file type. In particular, I believe the most troubling are the newer Microsoft Office formats because

they are compressed which complicates the algorithm locating the starting point of unique content. In these instances, it would be best to use a larger signature than the default 8 bytes. This should limit the occurrence of false positives but comes at the cost of a computationally more challenging signature. To test this hypothesis I repeated the previous experiment with a signature length of 16 bytes.

File Type	False Positives
<b>.txt</b>	0
<b>.doc</b>	0
<b>.docx</b>	0
<b>.exe</b>	0
<b>.pdf</b>	0
<b>.xls</b>	0
<b>.xlsx</b>	0

**Table 16:** Large Signature False Positives

This is in contrast to the results obtained in the experiments on optimal signature length. The optimal signature length assumed the data being processed was random; thus it would only be by chance that the signatures would match. When compared to real world data that is in another format this assumption held. However, when comparing content with the same format the occurrence of false positives is much higher and requires a longer signature length. However this was only true across some of the file formats so increasing the default signature length was not necessary. Instead a larger signature length

can be manually specified when dealing with files known to cause a higher rate of false positives.

### **Signature Validity**

While I verified that the signature was being generated and it worked within my environment, I wanted to take extra precaution that the signatures generated were valid. To perform this test I created a rules file with all the signatures from the signature matching and parsed it using a tool called DumbPig<sup>9</sup>. DumbPig is a tool that belongs to the SourceFire, and it currently is under development by Leon Ward. This tool not only checks for bad grammar or bad syntax which would result in a failure of Snort to launch, but it also checks for possible performance-intense rules. This test in particular highlighted the performance problems of using the default configuration that specifies any source and any destination over any protocol. The application is called from the command line with a flag specifying the rules file:

```
>./dumbpig -r allRules.rules
```

While rules that use the default any for protocol, source and destination are performance problems, I was not able to identify a more suitable default value. The onus of creating good rules with enough variables to filter the processing to an acceptable level will be placed on the users.

### **Random Distribution**

To work around the commonalities between files that share the same starting point I created the ability to randomize the starting point by including the random flag at startup time. The default starting location is at the middle of the hex String file, and this location

is either incremented or decremented by 25% resulting in a potential starting location of 25-75% of the file String length.

To test this I selected a single file (randText0.txt) and ran it through the algorithm with the -r flag 20 times using the command:

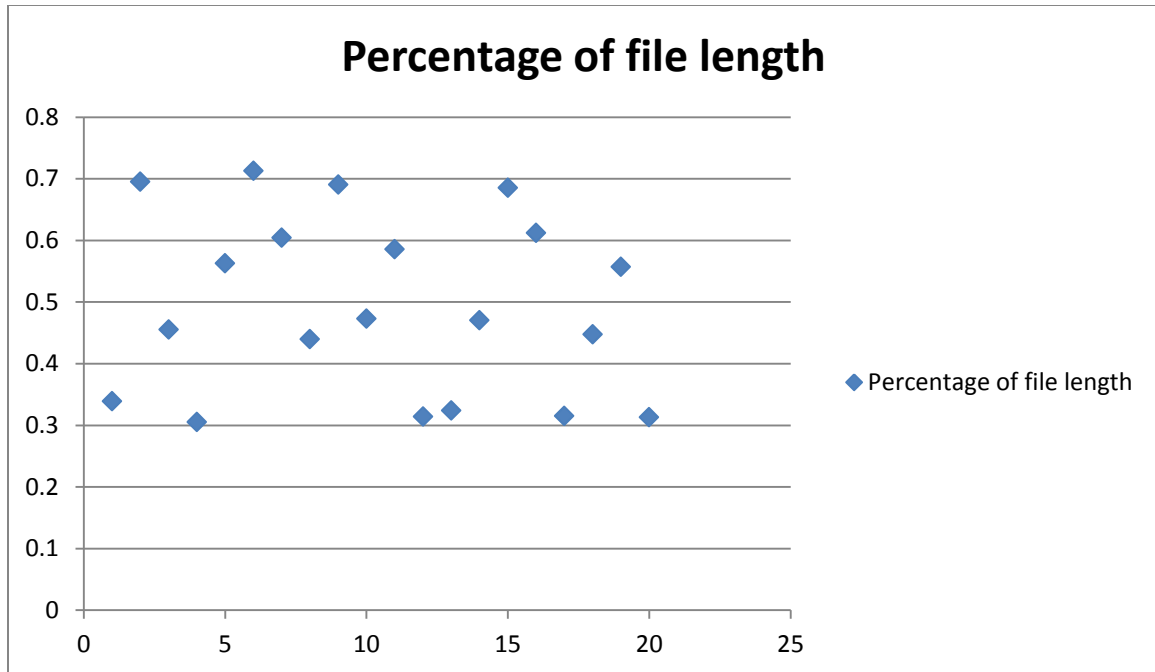
```
>python autoSnortSig.py -f randText0.txt -r
```

Which generated the output:

```
>alert any any any -> any any (content: "|6c61476273365556|";  
>msg: "randText0.txt";)
```

As a control, I included the original signature "4c6c5b326c7c346e," which should have a starting point at the middle of the file since the text files are under 10mb. Then, I determined the starting point using the algorithm within randCheck.py, found in Appendix B, to find the percentage of the file's length at which the signatures were found. That data is shown graphically in the **Figure 6** below.





**Figure 6:** Percentage of file length

Analyzing the data, the starting points are spread uniformly between the values 0.25 and 0.75 which is the intended result. The smallest value was 0.3130 and the largest value was 0.7130 which shows the results spanning the entire spectrum of intended starting points.

## **Conclusion & Future Work**

It is possible to create signatures for files in automated fashion, but they have inherent flaws that leave them inferior to manually created signatures. Most noticeable is the fact that some file types have content that is not unique to the file in question. This would result in a signature that could identify the file and would not cause an abundance of false positives when compared against random data; however, it would result in false positives when analyzing files of the same extension. This can be compensated for by using a larger signature or by altering the algorithm to take specific measures based on the file extension. The biggest challenge, however, was attempting to develop a single algorithm which was applicable to all file types. The current algorithm provided no false negatives while testing but it is susceptible to false positives across the same file types. Digging deeper into the file type with the highest false positive rate, Word documents, led to superior results at a larger signature size. It is clear that each file type would have superior results by altering the starting location (or using the randomization function) as well as using a custom signature size. However, the default behavior of this algorithm was designed to address an unknown file type; therefore, it is currently configured to use the settings that lead to good signatures across many file types.

Going forward I believe additional research can be done into the various file formats to identify a custom algorithm that limits the number of false positives for files of the same type. For files stored in a binary format, this may be trivial or even unnecessary.

However, for more complex file formats the process of identifying the start of unique content could be challenging. In particular, the new Microsoft Office file format would require a unique approach allowing the beginning of the document's unique content to be

identified when it is unencrypted and then again identify the start of that same content in the resulting encrypted file. In these cases, it may be best to not look for a set starting point, but rather choose a starting point based on a set offset or a certain percentage of file size. These settings could be tested by file type to determine best practices and implemented into the algorithm which already determines the file type that is being processed.

Finally, Snort has the ability to perform various actions based on the conditions of the rule being met. At this time, the algorithm only supports alerting when the conditions are met but scenarios may arise where other actions would better serve the end user's needs. I would like to add this option as another optional argument.

### **Signature Generation**

The signature generation process currently employed takes a broad stroke with no special processing dependent on the type of file being processed. This shows that an algorithm can be used to create signatures in an automated manner. However I believe further research can be done into popular data types to determine what methodologies work best. Each data type may be better served by a unique signature size as well as a unique starting point so that the unique data of that data type is best collected by the algorithm. This would result in signatures less likely resulting in false positives due to signatures being pulled from areas common to multiple files within a data type.

### **Service**

I believe that the algorithm would provide great value, if it could be run as a service and set to monitor a directory or directories. When it notices a new or changed file, it would update a rules file accordingly. To do this, the service would need to track the files by

storing values in memory or keeping a log file containing entries for each file. I believe the best way to track the files would be to create an entry per line containing the file name, signature content, last modified date, and hash value. With this data set, it would be possible to quickly check last modified date or if a new file exists while only hashing all the files at a separate longer interval. Additionally the service would have access to a larger data set since it will have a history of past runs and can be trained to identify signatures that are shared across multiple files and identify them as bad. By expanding the data set and establishing a history the algorithm can then begin to create better signatures in an automated manner.

### **Script**

As an alternative to operating the algorithm as a service, operation can also be configured as a recurring script (i.e. through a schedule task or a cron job). This process would not be difficult, but it would require a separate rules file where the output overwrote the files content each run. This would provide another way to automate the updating of a single file or a directory's signature generation. This is ideal for a server, such as an FTP server, whose contents can update without an administrator's knowledge. These files could be protected without any administrator actions being taken, once the recurring script was configured.

### **Additional Parameters**

The Snort rule set allows for additional customization not available for automation in the algorithm's current version. I have included all elements necessary to create a solid rule set, but additional options create additional value for a small subset of users. I would like the direction of these additional parameters to be need-driven based on user feedback. A

good example of an additional parameter is the directionality of the rule. Currently the directionality is defaulted to “source -> destination” but Snort rules also allow a bi-directional operator “<->” and the reverse direction “<-“. An additional parameter could be accepted at runtime allowing for the directionality to be specified as well as other parameters.

Two parameters which could create better performing rules are the offset and depth flags within the content. These rules give more specificity to Snort as to where to search for the content that is chosen as the signature. By being more specific the computational costs of the signatures are reduced and this allows for more rules to be used within the same environment.

## **Integration**

The Snort community has a large list of add-ons developed either in house, by SourceFire, or by outside companies or individuals. This algorithm could be included in those offerings or integrated into some of the products they already offer. I believe that this tool would be an ideal candidate for an open source community, as they could identify and pursue additional file types ensuring file types used are most likely to have the best support.

## **GUI**

A graphical front end would simplify the process of using the algorithm. Python toolkits such as Tkinter and wxPython exist to allow for GUI to be constructed in 2D or even 3D. The back end functions have been designed to tap into a GUI with the ability to provide superior feedback or even generate the signature on the fly as the user updates his or her entries. Since the goal of this research was to simplify and automate the Snort rule

generation process, I believe that going a step further and creating a GUI is something that should be pursued in the future.

### **Noise Traffic**

In the current experiments the packet captures are limited to either randomized data or a single protocol (FTP). Additional testing should be done using packet captures that contain the element being searched for as well as other traffic that would emulate a typical networking environment. This would identify if the algorithm is creating rules that can differentiate between the intended content and all the additional noise. While the signatures were tested against just noise traffic and against just the intended content there is a possibility of the combined traffic producing unintended results. These tests should be performed to eliminate the possibility of difficulties when the algorithm is implemented into a real world environment where the intended content and noise will be within a single data set.

### **Summary**

Information security is a tricky business. It takes only one vulnerability for an attacker to exploit and compromise the entire system's integrity. Additionally, large data leaks are done by employees or other privileged users. This tool is designed to build on the success of Snort and empower administrators to control the flow of their valuable information. It is possible to generate signatures for Snort rules in an automated fashion across a file or an entire directory. The process is quick and requires little knowledge of how the rule's content is generated. The algorithm takes in content and parses it making limited decisions on how to generate a valid signature. This process has been shown to work and

is a starting point for turning Snort into a content protection system that limits data leakage.

By enabling protectors to be proactive, they can have the free time to better understand their unique needs and adapt their technology accordingly. Security is often referred to as a game of cat and mouse where the attackers are always on the forefront of exploits and administrators are racing to stop the latest exploit. Without the ability to detect and stop zero day attacks that scenario may never change. To compensate a defense in depth approach must be taken so that multiple layers of defense stand between an attacker and the valuable information or services being hosted. This algorithm is another tool that can provide administrators further ability to protect key files or directories from traveling certain directions across their network.

The primary strength of this algorithm is that it can be applied to any file type, and the fact that it generates a valid, tested signature to match the content of the parsed file. This conclusively shows that an algorithm can be used to parse files and output Snort signatures.

The greatest weakness of this approach is that it takes a non-specific approach to files dependent on their file type. This leads to scenarios where a better signature could be generated by using an algorithm tuned to a specific data type. I believe that this is where the greatest future work of this algorithm lies, and that further research may be done for each file type to determine the best case starting point and signature length independent of file size.

**Appendix A: Source Code – autoSnortSig.py**

```
#!/usr/bin/python
#Designed to read in files and print out snort signatures
#Requires Python 3.3
#
#
#JMU Thesis Program
#Computer Science
#ricebf@dukes.jmu.edu
#
#
#Coded by: Brandon Rice
#Last modified 6/23/2014
```

```
import sys, argparse, binascii, os, random
from os.path import isfile, join
```

```
#Begin global variables
file = ""
fileString = ""
source = "any"
destination = "any"
sourcePort = ""
destinationPort = ""
protocol = "any"
matchSignature = ""
signature = ""
size = 8
fileExtension = ""
verbose = False
```



```

directory = ""
rand = False

#Begin Classes
#main executes the core logic
def main():
    global file
    global directory
    argRead()
    #Determine if a file or a directory needs to be parsed
    if file == "" and directory == "":
        print("Either a directory or a file must be specified.")
    elif file != "" and directory != "":
        print("File and directory are exclusive options. Only one may be enabled at a time.")
    elif file != "":
        fileParse(file)
    else:
        dirParse()

#Function to process a single file
def fileParse(toDo):
    global fileString
    global fileExtension
    global file
    file = toDo
    vPrint("File to be read " + file)
    vPrint("File extension: " + fileExtension)
    fileRead()
    #test if file is being read in properly
    #print(fileString)
    ipCheck()
    validSize()

```

```

validProtocol()
createSig()
combineSig()
print(signature)

#Function to parse a directory calling all filse within the directory
def dirParse():
    global directory
    print("Processing directory: " + directory)
    #Create a list of files to be parsed
    files = [f for f in os.listdir(directory) if isfile(join(directory,f)) ]
    vPrint(files)
    for x in files:
        #Parsing each file requires path + file name
        fileParse(directory + x)

#Function to print certain lines only if verbose is True
def vPrint(x):
    global verbose
    if verbose == True:
        print(x)

#argRead() takes in the args and prints help if needed
def argRead():
    global file
    global source
    global destination
    global protocol
    global size
    global fileExtension
    global verbose

```

```

global directory
global rand

parser = argparse.ArgumentParser(description="AutoSnortSignature")
parser.add_argument("-f", "--file", help="Input file name", required=False)

parser.add_argument("-s", "--source", help="Source IP (port optional). Format example
192.168.100.1 or 192.168.100.1:22", required=False)

parser.add_argument("-d", "--destination", help="Destination IP (port optional). Format
example 192.168.100.1 or 192.168.100.1:22", required=False)

parser.add_argument("-p", "--protocol", help="Specify the protocol. Acceptable input IP,
TCP, UDP, ICMP", required=False)

parser.add_argument("-l", "--length", help="Specify the signature length in number of
bytes (1-64)", required=False)

parser.add_argument("-v", "--verbose", help="Enable verbose output
mode", required=False, action='store_true')

parser.add_argument("-a", "--all", help="Parse all files within the given
directory", required=False)

parser.add_argument("-r", "--random", help="Randomize starting point to provide
additional security", required=False, action='store_true')

args = parser.parse_args()

if args.file is not None:
    file = args.file
    temp, fileExtension = os.path.splitext(file)

if args.source is not None:
    source = args.source

if args.destination is not None:
    destination = args.destination

if args.protocol is not None:
    protocol = args.protocol

if args.length is not None:
    size = int(args.length)

if args.verbose is not False:
    verbose = True

if args.all is not None:
    directory = args.all
    directory = directory.replace("\\", "/")

```

```

    if directory[-1] != '/':
        directory = directory + '/'
    if args.random is not False:
        rand = True

#Reads in the provided file and stores it as a String of hex characters
def fileRead():
    global fileString
    global file
    with open(file, "rb") as f:
        content = f.read()
    fileString = binascii.hexlify(content)
    #Limit maximum portion of file to be processed to 10mb
    if (len(fileString) > 2621440):
        fileString = fileString[:2621440]

#Validates source and destination if set otherwise default to any
def ipCheck():
    global source
    global destination
    global sourcePort
    global destinationPort
    #Extract the port from the source
    where = source.find(":")
    #print("Source where is " + str(where)) #Enable for diagnostics
    if where == -1:
        sourcePort = "any"
    else:
        sourcePort = source[(where+1):]
        source = source[:where]
    #Extract the port from the destination
    where = destination.find(":")

```

```

#print("Source where is " + str(where)) #Enable for diagnostics
if (where > -1):
    destinationPort = destination[(where+1):]
    destination = destination[:where]
else:
    destinationPort = "any"
#Code here to validate the ports
if sourcePort != "any":
    vPrint("Validating source port")
    validPort(sourcePort)
if destinationPort != "any":
    vPrint("Validating destination port")
    validPort(destinationPort)
#Code here to validate the IP addresses
if source != "any":
    vPrint("Checking source IP")
    validIP(source)
if destination != "any":
    vPrint("Checking destination IP")
    validIP(destination)
#Print values to be used. Can later be disabled
vPrint("Source IP is " + source)
vPrint("Source port is " + sourcePort)
vPrint("Destination IP is " + destination)
vPrint("Destination port is " + destinationPort)

#Check whether a port is numeric and within the correct range (0-65535)
def validPort(port):
    if port.isdigit() == True:
        num = int(port)
        #test = (-1 < num)
        #print(test)
        if -1 < num and \

```

```

        num < 65536:
            return True
    else:
        print("Invalid port found. Exiting")
        sys.exit(2)
else:
    print("Invalid port found. Exiting")
    sys.exit(2)

#Check whether a valid IP address is provided
def validIP(ip):
    if ip.count(".") != 3:
        print("Invalid IP address found. Exiting")
        sys.exit(2)
    vPrint("Checking octave 1")
    where = ip.find(".") #Check first octave
    if (where > -1):
        if (validOctave(ip[:where]) == False):
            print("Invalid IP found")
            sys.exit(2)
        else:
            ip = ip[where+1:]
    else:
        print("Invalid IP found")
        sys.exit(2)
    vPrint("Checking octave 2")
    where = ip.find(".") #Check second octave
    if (where > -1):
        if (validOctave(ip[:where]) == False):
            print("Invalid IP found")
            sys.exit(2)
        else:
            ip = ip[where+1:]

```

```

else:
    print("Invalid IP found")
    sys.exit(2)
vPrint("Checking octave 3")
where = ip.find(".") #Check third octave
if (where > -1):
    if (validOctave(ip[:where]) == False):
        print("Invalid IP found")
        sys.exit(2)
    else:
        ip = ip[where+1:]
else:
    print("Invalid IP found")
    sys.exit(2)
vPrint("Checking octave 4")
if (validOctave(ip) == False): #Check fourth octave
    print("Invalid IP found")
    sys.exit(2)

#Check if provided input is a valid octave
def validOctave(octave):
    if octave.isdigit() == True:
        if (-1 < int(octave) < 256):
            #print("Returning true on octave check")
            return True
        else:
            print("Invalid octave found. Exiting")
            sys.exit(2)
    else:
        print("Invalid octave found. Exiting")
        sys.exit(2)

#Check if the size is between 1 and 64. If not set the size to the default 8.

```

```

def validSize():
    global size
    vPrint("Size is: " + str(size))
    if 3 < size and size < 65:
        vPrint("Valid signature size found")
        return True
    else:
        print("Invalid size found setting size to the default")
        size = 8

#Check if a valid Snort protocol was specified. If not alter protocol to any
def validProtocol():
    global protocol
    acceptable = ["ip","tcp","udp","icmp","any"]
    protocol = protocol.lower()
    if protocol not in acceptable:
        protocol = "any"
        print("Invalid protocol value detected. Setting to the default any")

#Reads the file contents and stores the hex string that needs matching
#Starts at the middle of the file and pulls hex strings until one is found with an acceptable
amount of empty (0) values
def createSig():
    global matchSignature
    global fileString
    global size
    global rand
    #copy hex string for local manipulation
    temp = fileString
    #variable to store the hex pairs that represent a byte
    fileHexArray=[]
    #populating the array
    while temp.__len__() > 0:

```



```

        fileHexArray.append(temp[:2])
        temp = temp[2:]
    length = len(fileHexArray)
    start = int((length/2) - (size/2))
    #Check if random is true
    #If true alter the starting point by placing it in a random location from 25% to 75% of
    file length
    if rand == True:
        random.seed
        #Create variable for 0 to 25% file length
        alter = random.randint(0,start//2)
        #Either add or subtract the variable from start
        add = random.randint(0,1)
        if add == 0:
            start = start + alter
        else:
            start = start - alter
    matchSignature = ""
    #extracting the signature
    for i in range(size):
        temp = fileHexArray[start + i]
        temp = str(temp)
        temp = temp[2:4] #stripping the b' and ' characters leaving only the two hex
        characters
        matchSignature = matchSignature + temp
    #Determine the occurrence of 0 in the signature. If greater then 50% loop through the
    hex string searching
    #for a signature that has few enough 0 to be unique. Designed to prevent signatures
    that are padding.
    count = matchSignature.count("0")
    while (count > (size//2)):
        start = start + size
        if (start + size) > length:
            print("Unable to find a suitable string. Try decreasing the string size")

```

```

        sys.exit(2)
    matchSignature = ""
    for i in range(size):
        temp = fileHexArray[start + i]
        temp = str(temp)
        temp = temp[2:4]
        matchSignature = matchSignature + temp
    count = matchSignature.count("0")

```

#Combines all the signature elements into a valid signature string

```

def combineSig():
    global signature
    global protocol
    global source
    global sourcePort
    global destination
    global destinationPort
    global matchSignature
    global file
    signature = "alert "
    signature = signature + protocol + " "
    signature = signature + source + " "
    signature = signature + sourcePort + " -> "
    signature = signature + destination + " "
    signature = signature + destinationPort + " "
    signature = signature + "(content:\"|" + matchSignature + "\"\");"
    signature = signature + " msg:\"|\" + file + "\"\");"

```

#Begin application by calling main

```

main()

```

## Appendix B: Supplementary Code

### **printHex.py**

```
##print file as hex

import binascii
filename = 'randText0.txt' ##Must be manually changed
with open(filename, 'rb') as f:
    content = f.read()
print(binascii.hexlify(content[0:]))
```

### **genText.py**

```
##Prints a string of 1000 random ASCII printable characters

from random import choice
import string

def genText(length=8, chars=string.printable + string.digits):
    return ".join([choice(chars) for i in range(length)])

print(genText(1000,string.printable))
```

### **randCheck.py**

```
##Check validity of randomness
##Open randText0.txt and then find where the signatures
##are located within the file as a percentage of total length

import binascii

signatures = ["6c61476273365556", "5c363e316572274b",
              "6f54402b43724a74", "4d2d476c21375d61",
              "6f4d556a7b70477e", "5d733f3d4c5a6e66",
              "4f4050794253403a", "0d0a595672653474",
              "0a46562f245c363e", "70567e0b224a2b49",
              "4b56334c7247402f", "5834510c70375359",
              "7a462c5b59547727", "59696c70567e0b22",
              "474e38750d0a4656", "54652f36556f473b",
              "34510c703753595a", "380d21477c724068",
              "3d3a7c6e38636f4d", "235834510c703753"]

with open("randText0.txt", "rb") as f:
    content = f.read()
fileString = binascii.hexlify(content)
```

```
fileString = str(fileString)
length = len(fileString)

for sig in signatures:
    temp = fileString.find(sig)
    print("File found at: " + str(temp/length))
```

## Bibliography

1 – SourceFire (2014 April) *Snort.Org*. Retrieved from <https://www.snort.org/snort>

SourceFire (2014, April 17th). Retrieved from <https://www.snort.org/snort>

2 – Santos, Alysson (2009) *Automatic Signature Generation*

Santos, A. (2009). Federal University of Pernambuco: Automatic Signature Generation.

Retrieved from <http://www.cin.ufpe.br/~tg/2009-1/afs5.pdf>

3 – Zhang, Jiong. (2005) *Network Intrusion Detection Systems Using Random Forests Algorithm*

Zhang, J. (2005). Ann Arbor: Network Intrusion Detection Systems Using Random

Forests Algorithm. Retrieved from ProQuest Digital Dissertations. (304952174)

<http://search.proquest.com/docview/304952174?accountid=11667>

4 – Yu, Zhenwei (2007) *An Adaptive Automatically Tuning Intrusion Detection System*

Yu, Z. (2007). DAI-B 68/12, Jun 2008: An Adaptive Automatically Tuning Intrusion

Detection System. Retrieved from ProQuest Digital Dissertations. (304724547)

<http://search.proquest.com/docview/304724547?accountid=11667>

5 – Yu, Lu (2012) *Stochastic Tools for Network Security: Anonymity Protocol Analysis and Network Intrusion Detection*

Yu, L. (2012). DAI-B 74/05(E), Nov 2013: Stochastic Tools for Network Security.

Retrieved from ProQuest Digital Dissertations. (1285214973)

<http://search.proquest.com/docview/1285214973?accountid=11667>

6 – Massicotte, Frederic (2010) *A set of approaches to evaluate and address the accuracy problem in intrusion detection systems*

Massicotte, F. (2010). DAI-B 73/04, Oct 2012: A set of approaches to evaluation and

address the accuracy problem in intrusion detection systems. Retrieved from

ProQuest Digital Dissertations. (917251459)

<http://search.proquest.com/docview/917251459?accountid=11667>

7 – Fugate, Sunny (2012) *Methods for Speculatively Bootstrapping Better Intrusion Detection System Performance*

Fugate, S. (2012). DAI-B 74/06(E), Dec 2013: Methods for Speculatively Bootstrapping

Better Intrusion Detection System Performance. Retrieved from ProQuest Digital

Dissertations. (1315234251)

<http://search.proquest.com/docview/1315234251?accountid=11667>

8 – Ward, Leon (June 2009) *DumbPig | An alchemists view from the bar*, Retrieved from <http://leonward.wordpress.com/dumbpig/>

DumbPig | An alchemist's view from the bar. (2009, June 10th). Retrieved from <http://leonward.wordpress.com/dumbpig/>

9 – Python Software Foundation (2014) *Welcome to Python!*, Retrieved from <https://www.python.org/>

Welcome to Python! (2014, July 17th). Retrieved from <https://www.python.org/>